lab4-stats

November 12, 2024

1 Time series forecasting

[1]:
```
import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)
warnings.simplefilter(action="ignore", category=UserWarning)

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

We will use sktime as our main library for time series. It offers interface very similar to scikit-learn, and conveniently wraps many other libraries, for example: - statsforecast - efficient implementations of many forecasting methods, e.g. AutoARIMA and AutoETS - pmdarima - statistical tests for time series and another AutoARIMA implementation - statsmodels - a few time series decomposition and forecasting methods

For statistical tests we will use scipy and statsmodels.

1.1 Forecasting Polish inflation

The problem of forecasting inflation (here defined using consumer price index, CPI) is very common, done by basically every country and larger financial institutions. In practice it's not a single task, but rather a collection of related problems, forecasting e.g. inflation, core inflation (excluding most volatile components, e.g. food and energy prices), and other formulations.

In Poland, basic data about inflation is published by the Central Statistical Office of Poland (GUS), with monthly, quarterly, half-yearly and yearly frequency. More detailed information is published by other institutions, because they depend on the methodology used, e.g. core inflation is calculated and published by the National Bank of Poland (NBP).

Forecasting inflation is a challenge, since it typically: - has visible cycles, but very irregular - is implicitly tied to many external factors (global economy, political decisions etc.) - there is no apparent seasonality - we are interested in forecasting with many frequencies, e.g. monthly (short-term decisions) and yearly (long-term decisions)

We will use GUS data with monthly frequency. To get a percentage value (annual percentage rate inflation) from the raw data, we need to subtract 100 from provided values.

[2]:
```python
y = pd.read_csv("polish_inflation.csv")
y = y.rename(columns={"Rok": "year", "Miesiąc": "month", "Wartość": "value"})

# create proper date columna
y["day"] = 1
y["date"] = pd.to_datetime(y[["year", "month", "day"]])
y["date"] = y["date"].dt.to_period("M")

# set datetime index
y = y.set_index(y["date"], drop=True)
y = y.sort_index()

# leave only time series values
y = y["value"] - 100

# filter out NaN values from the end of the series
y = y[~y.isna()]

y
```

[2]:
```
date
1982-01     53.2
1982-02    106.4
1982-03    110.7
1982-04    104.1
1982-05    108.4
           …
2024-04      2.4
2024-05      2.5
2024-06      2.6
2024-07      4.2
2024-08      4.3
Freq: M, Name: value, Length: 512, dtype: float64
```

To plot the time series, the easiest way is to use the plot_series() function from sktime, which will automatically nicely format X and Y axes.

[3]:
```python
from sktime.utils.plotting import plot_series

plot_series(y, title="Polish inflation")
```

[3]:
```
(<Figure size 1600x400 with 1 Axes>,
 <Axes: title={'center': 'Polish inflation'}, ylabel='value'>)
```

There is no error here - 90s were a particularly interesting period, with hyperinflation, later "shock therapy" and implementation of the Balcerowicz Plan. From the perspective of time series forecasting, this is definitely na outlier, but quite long. For this reason, we will limit ourselves to post-2000 data.

Similar behavior can often be seen in time series data, related to e.g. 2007-2008 financial crisis or COVID-19 pandemic. Such events can introduce shocks with long effects, and using only later data is arguably the simplest strategy to deal with this.

[4]:
```
y = y[y.index >= "2000-01"]
plot_series(y, title="Polish inflation, from year 2000")
```

[4]:
```
(<Figure size 1600x400 with 1 Axes>,
 <Axes: title={'center': 'Polish inflation, from year 2000'}, ylabel='value'>)
```

There is definitely some information here, with cycles and trends. Fortunately, the data seems to be changing reasonably slowly most of the time. But what about seasonality?

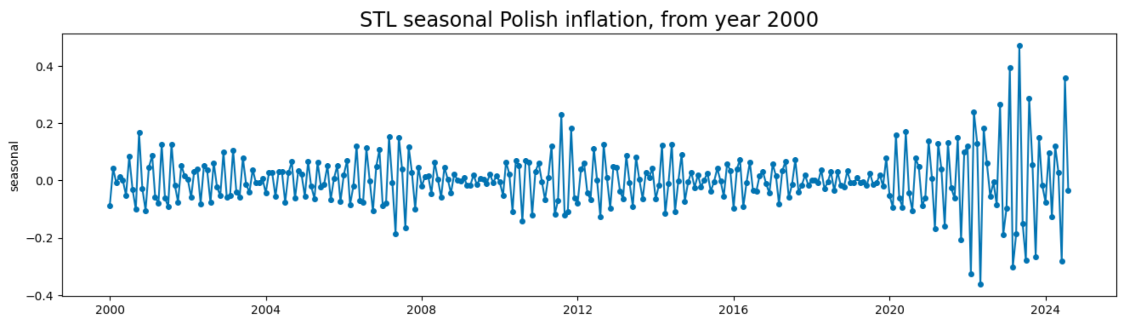
Exercise 1 (0.5 points)

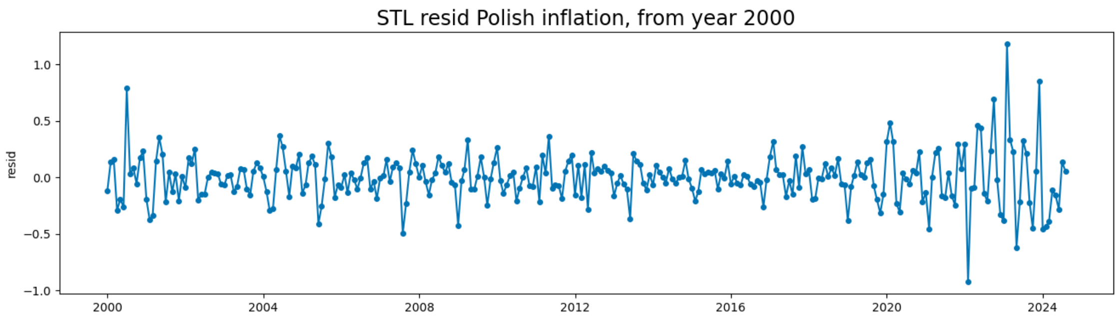
Implement the `plot_stl_decomposition` function. Use `STLTransformer` to compute the STL decomposition (documentation). Remember to use appropriate arguments to set the seasonality period and return all three components.

Plot the resulting STL decomposition. Comment: - do you see a yearly seasonality here? - concerning residuals, are they only a white noise, or do they seem to contain some further information to use?

[5]:
```
from sktime.transformations.series.detrend import STLTransformer
from sktime.datasets import load_airline

def plot_stl_decomposition(data: pd.Series, seasonal_period: int = 12) -> None:
    transformer = STLTransformer(sp=seasonal_period, return_components=True)
    Xt = transformer.fit_transform(data)
    # return Xt, transformer
    plot_series(Xt[['trend']], title="STL trend Polish inflation, from year␣
↪2000")
    plot_series(Xt[['seasonal']], title="STL seasonal Polish inflation, from␣
↪year 2000")
    plot_series(Xt[['resid']], title="STL resid Polish inflation, from year␣
↪2000")
```

[6]:
```
plot_stl_decomposition(y, seasonal_period=3)
```





2 // commented

There is some yearly seasonality, as there is low at the bigning of year (plut shome shift), top in the middle and again low at the end. Residual seems to be rather white noise

Manual check using STL decomposition is useful - this allows us to gain intuition and knowledge about the data, and validation parameters. Of course we also have automated procedures, using statistical tests, to avoid such manual labor when we can.

Let's check the seasonality and stationarity of our data. This is not strictly necessary for ETS models - they use the data as-is. However, the ARIMA models require stationary data, and knowledge about seasonality, or lack thereof, can greatly accelerate our experiments. SARIMA takes much longer than simpler ARIMA.

Exercise 2 (0.75 points)

1. Check, using statistical tests for seasonality, if there is a quarterly, half-yearly, or yearly seasonality in the data. Use the `nsdiffs` function from pmdarima (documentation). If you detect seasonality, remove it using the `Differencer` from sktime (documentation) and plot the deasonalized series.
2. Check, using statistical tests for stationarity, what differencing order stationarizes the data. Use the `ndiffs` function from pmdarima (documentation). If it's greater than zero, i.e. differencing is necessary, then stationarize the series using the `Differencer` class and plot the resulting time series.
3. Comment, which ARIMA model would you use, based on those findings, and why: ARMA, ARIMA, or SARIMA.

Use the default `D_max` and `d_max` values.

Warning: create new variables for values after differencing, do not overwrite the `df` variable. It will be used later.

[7]:
```
from pmdarima.arima import nsdiffs, ndiffs
from sktime.transformations.series.difference import Differencer

def find_and_remove_seasonality(df, m):
```

```
    if nsdiffs(df, m):
        transformer = Differencer(lags=m)
        return  transformer.fit_transform(df), True
    return df, False
```

[8]:
```
m = 3
diff_df, processed = find_and_remove_seasonality(y, m)
print(f'Seasonality for period {m}', processed)

m = 6
diff_df, processed = find_and_remove_seasonality(diff_df, m)
print(f'Seasonality for period {m}', processed)

m = 12
diff_df, processed = find_and_remove_seasonality(diff_df, m)
print(f'Seasonality for period {m}', processed)
```

```
Seasonality for period 3 False
Seasonality for period 6 False
Seasonality for period 12 False
```

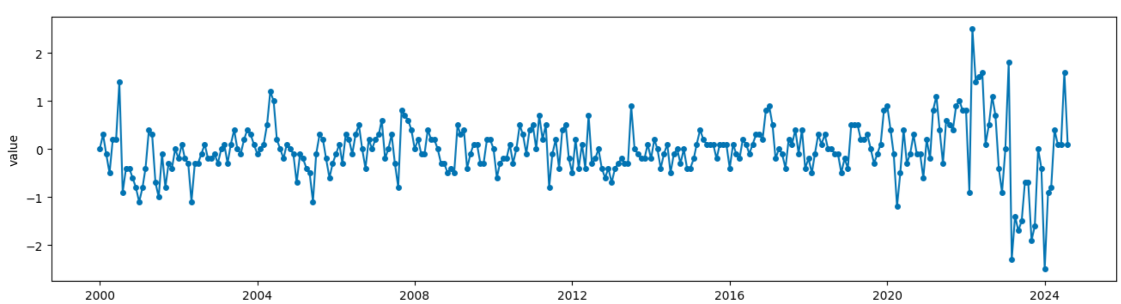
[9]:
```
def find_and_remove_trend(df):
    if ndiffs(df):
        transformer = Differencer(lags=1)
        return  transformer.fit_transform(df), True
    return df, False
```

[10]:
```
diff_df, processed = find_and_remove_trend(diff_df)
print('Removed trend', processed)
```

```
Removed trend True
```

[11]:
```
plot_series(diff_df)
```

[11]:
```
(<Figure size 1600x400 with 1 Axes>, <Axes: ylabel='value'>)
```



3 // commented

Tests' results have shown that there is trend but not seasonality, so ARIMA model would be first choice.

We are now basically ready to train our forecasting models. We will use 20% of the newest data for testing, using the expanding window strategy, with step 1 (we get inflation reading each month). MAE and MASE will be used as quality metrics.

We will also perform residuals analysis. Errors should be normally distributed (unbiased model) and do not have autocorrelation (model utilizing all available information). For all statistical tests we assume the significance level $\alpha = 0.05$.

For testing normality, the Anderson-Darling test is less conservative than Shapiro-Wilk test, which is quite useful in practice. Errors are very rarely close to "true" normality in real world. The null hypothesis is that values come from the given distributions (by default the normal one), and alternative hypothesis that they come from other distribution.

For testing error autocorrelation, the Ljung-Box test is used, which tests autocorrelation for various lags. For each lag, a separate test is performed. The null hypothesis is the lack of autocorrelation, and the alternative hypothesis is that there is an autocorrelation with a given lag.

Exercise 3 (1.5 points)

Implement the missing parts of the `evaluate_model` function: 1. Create `ExpandingWindowSplitter` (documentation), which should start testing at 80% of data. The forecast window size is controlled via the `horizon` parameter. 2. Create a list of metric objects, consisting of MAE and MASE (ocumentation). 3. Perform the model evaluation, using the `evaluate` function (documentation). Pass `return_data=True`, in order to also return the computed forecasts. It returns a DataFrame with results. 4. Calculate average metric values, using the resulting DataFrame. Print them rounded to 2 decimal places. 5. Taking into consideration the `analyze_residuals` argument, perform the error analysis: - calculate residuals $y - \hat{y}$ - plot the residuals histogram - perform the Anderson-Darling test (documentation) and print whether the distribution is normal or not - perform the Ljung-Box test (documentation) and print the test results

Test the function, using two baseline forecasting methods: average (mean) and last known value. Use the `NaiveForecaster` class (documentation), with 3 months forecasting horizon. Plot the forecasts, using the `plot_forecasts` argument.

[12]:
```
from joblib import Parallel, delayed
import pandas as pd
from scipy.stats import anderson
from sktime.forecasting.model_evaluation import evaluate
from sktime.performance_metrics.forecasting import (
    MeanAbsoluteScaledError,
    MeanAbsoluteError,
    MeanAbsolutePercentageError,
)
from sktime.forecasting.model_selection import ExpandingWindowSplitter
from statsmodels.stats.diagnostic import acorr_ljungbox
```

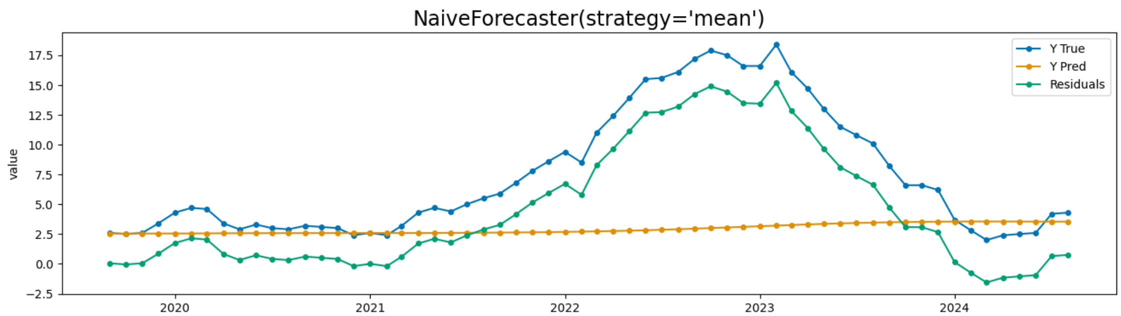
```python
# from sktime.utils.plotting.forecasting import plot_series
import matplotlib.pyplot as plt
from tqdm_joblib import tqdm_joblib
from tqdm import tqdm

# Define a custom function to evaluate on each fold
def evaluate_on_fold(model, train_data, test_data, fh, metrics):
    model.fit(train_data)
    y_pred = model.predict(fh)
    y_true = test_data.loc[y_pred.index]
    results = {metric.name: metric(test_data=test_data, y_pred=y_pred,␣
↪y_true=y_true, y_train=train_data) for metric in metrics}
    results['y_pred'] = y_pred
    return results

def evaluate_model(
    model,
    data: pd.Series,
    horizon: int = 1,
    plot_forecasts: bool = False,
    analyze_residuals: bool = False,
) -> None:
    # Define expanding window cross-validation with the forecast horizon
    cv = ExpandingWindowSplitter(fh=[i + 1 for i in range(horizon)],␣
↪initial_window=int(0.8 * len(data)), step_length=1)

    # Define the metrics to be calculated
    metrics = [
        MeanAbsoluteError(),
        MeanAbsoluteScaledError(),
    ]

    # Perform cross-validation
    # results = evaluate(model, cv=cv, y=data, strategy="refit",␣
↪return_data=True, scoring=metrics)

    with tqdm_joblib(tqdm(desc="Processing tasks", total=len([0 for _ in cv.
↪split(data)]))):
        results = Parallel(n_jobs=-1)(
            delayed(evaluate_on_fold)(model, data.iloc[train], data.iloc[test],␣
↪cv.fh, metrics)
            for train, test in cv.split(data)
        )

    results = pd.DataFrame(results)

    # Extract evaluation metrics
```

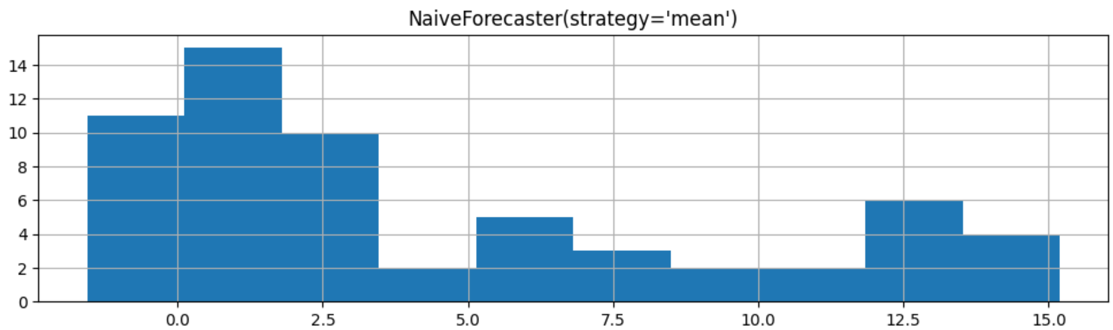
```python
    mae = results["MeanAbsoluteError"].mean()
    mase = results["MeanAbsoluteScaledError"].mean()

    # print(f"MAE: {mae}")
    # print(f"MASE: {mase}")

    print(f"MAE: {mae:.2f}")
    print(f"MASE: {mase:.2f}")

    # Concatenate predictions from each fold
    y_pred = pd.concat(results["y_pred"].values).sort_index().groupby(level=0).
 ↪mean()

    y_true = data.loc[y_pred.index]  # True values matching predicted indices

    # Optionally plot forecasts
    if plot_forecasts:
        # plot_series(, title="Y Pred")
        residuals = y_true - y_pred
        plot_series(y_true, y_pred, residuals, labels=["Y True", "Y Pred",␣
 ↪'Residuals'], title=str(model))
        plt.figure()
        residuals.hist(figsize=(12, 3))
        plt.title(str(model))

        plt.show()
        plt.clf()

    # Optionally analyze residuals
    if analyze_residuals:
        residuals = y_true - y_pred

        # Anderson-Darling test for normality of residuals
        anderson_result = anderson(residuals.values.reshape(-1))
        print("Anderson-Darling test statistic:", anderson_result.statistic)

        # Ljung-Box test for autocorrelation in residuals
        ljung_box_result = acorr_ljungbox(residuals, lags=[10], return_df=True)
        print("Ljung-Box test p-values:", ljung_box_result["lb_pvalue"].values)
```

[13]:
```python
from sktime.forecasting.naive import NaiveForecaster
```

[14]:
```python
# 1. Mean Forecasting
print("Testing Naive Forecaster with mean strategy")
mean_forecaster = NaiveForecaster(strategy="mean")
evaluate_model(mean_forecaster, data=y, horizon=3, plot_forecasts=True,␣
 ↪analyze_residuals=True)
```

```
Testing Naive Forecaster with mean strategy

Processing tasks:   0%|          | 0/58 [00:00<?, ?it/s]

  0%|          | 0/58 [00:00<?, ?it/s]

MAE: 4.88
MASE: 14.24
```






```
Anderson-Darling test statistic: 3.3942040126279096
Ljung-Box test p-values: [7.57493243e-59]

<Figure size 640x480 with 0 Axes>
```

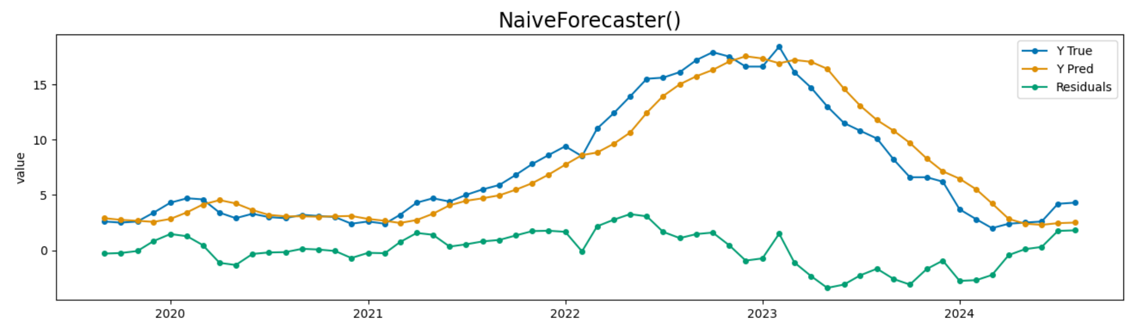
Here, first test results (greater than 0.05) allows to reject hypothesis that residuals follows normal distribution. (It's also visible on histogram) Also, based on the seconds test it can be said that there is correlation between strong residuals.

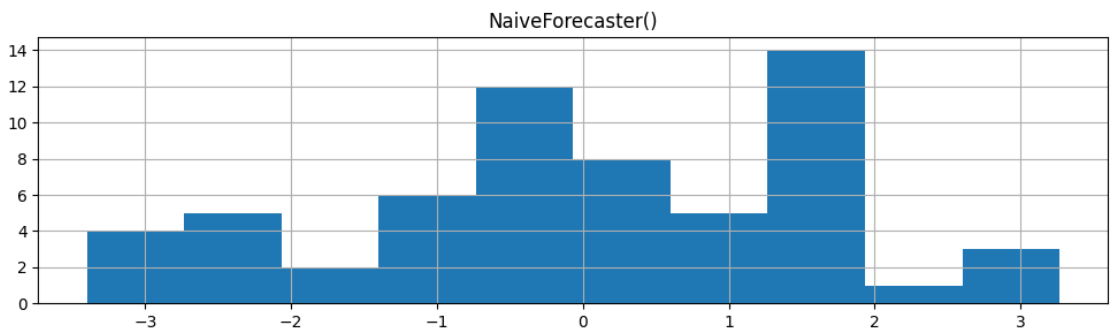
[15]:
```python
# 2. Testing Naive Forecaster with last known value strategy
print("\nTesting Naive Forecaster with last known value strategy")
last_forecaster = NaiveForecaster(strategy="last")
evaluate_model(last_forecaster, data=y, horizon=3, plot_forecasts=True,␣
 ↪analyze_residuals=True)
```

```
Testing Naive Forecaster with last known value strategy
```

```
  0%|          | 0/58 [00:00<?, ?it/s]
```

```
MAE: 1.33
MASE: 3.89
```






```
Anderson-Darling test statistic: 0.5817872501377295
Ljung-Box test p-values: [3.4507801e-23]
```

```
<Figure size 640x480 with 0 Axes>
```

Here we can assume, that residulas follo normal distribution. But still, are correlated.

Results from our first baselines look reasonable. Let's see how ETS and ARIMA will compare.

Exercise 4 (0.75 points)

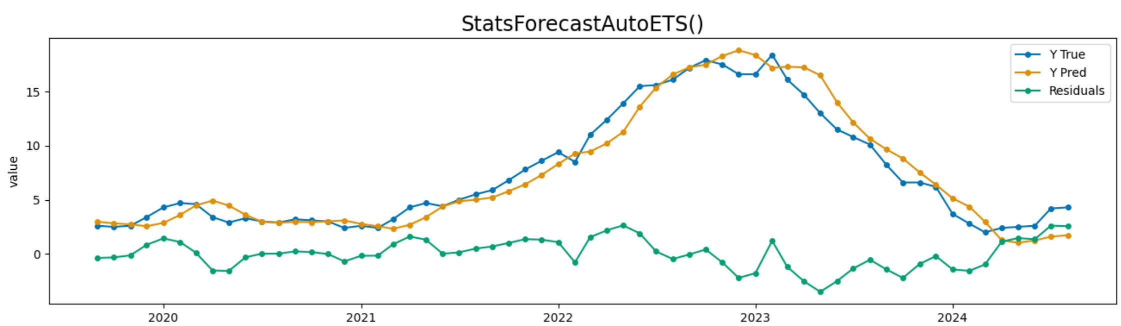
1. Perform forecasting using the AutoETS algorithm in the damped trend variant, based on the `statsforecast` implementation (documentation). Plot forecasts and perform residuals analysis.
2. Similarly, use AutoARIMA for forecasting (documentation). If you didn't detect seasonality earlier, pass appropriate option to ignore SARIMA variants.
3. Comment on the results:
   - did you manage to outperform the baselines?
   - which of the models is better, and what may this mean?

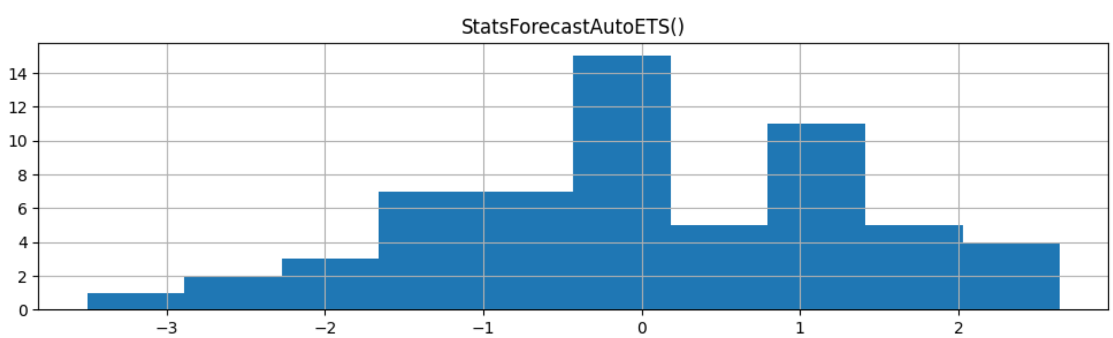
- which model is correct, at least approximately, i.e. has normally distributed, non-autocorrelated errors?
- are the results of the best model, subjectively, good enough?

As before, use 3 month forecast horizon.

[16]:
```
from sktime.forecasting.statsforecast import StatsForecastAutoETS
from sktime.forecasting.arima import AutoARIMA

# 1. Testing AutoETS Model with Damped Trend
print("Testing AutoETS with damped trend")
ets_forecaster = StatsForecastAutoETS(model="ZZZ") # "ZZD" denotes damped trend
evaluate_model(ets_forecaster, data=y, horizon=3, plot_forecasts=True,␣
 ↪analyze_residuals=True)
```

```
Processing tasks:   0%|          | 0/58 [00:03<?, ?it/s]
Processing tasks:   0%|          | 0/58 [00:00<?, ?it/s]

Testing AutoETS with damped trend


Processing tasks:   0%|          | 0/58 [00:00<?, ?it/s]

  0%|          | 0/58 [00:00<?, ?it/s]

MAE: 1.16
MASE: 3.39
```





```
Anderson-Darling test statistic: 0.27266606021447615
Ljung-Box test p-values: [2.29640363e-08]
```

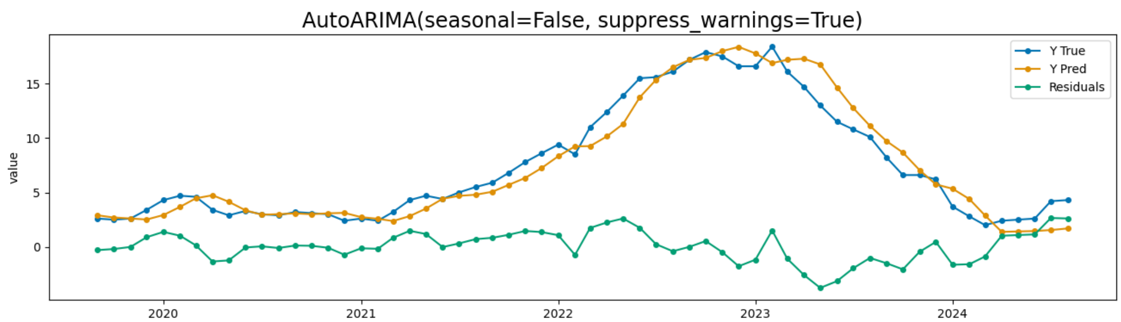
```
<Figure size 640x480 with 0 Axes>
```

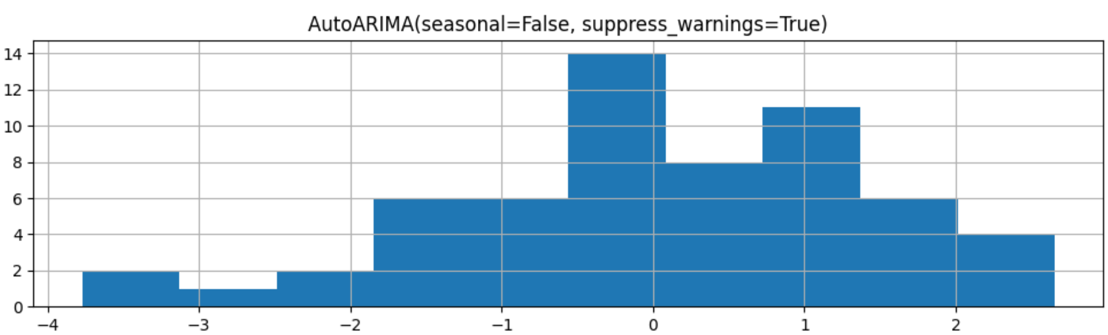
[17]:
```
# 2. Testing AutoARIMA Model without Seasonality
print("\nTesting AutoARIMA without seasonality")
arima_forecaster = AutoARIMA(suppress_warnings=True, seasonal=False)
evaluate_model(arima_forecaster, data=y, horizon=3, plot_forecasts=True,␣
↪analyze_residuals=True)
```

```
Testing AutoARIMA without seasonality
```

```
  0%|          | 0/58 [00:00<?, ?it/s]
```

```
MAE: 1.16
MASE: 3.40
```






```
Anderson-Darling test statistic: 0.3087261331452993
Ljung-Box test p-values: [5.10392521e-09]
```

```
<Figure size 640x480 with 0 Axes>
```

4 // commented

AutoETS and AuroARIMA gave slightly better results than Naive forecaster with strategy 'last'. 1. AutoETS: MAE: 1.33 MASE: 3.89 2. AutoARIMA: MAE: 1.16 MASE: 3.39 3. Naive (strategy - last): MAE: 1.33 MASE: 3.89

Bot non-trivial models gave quite the same results

Both models seems to give normally distributed errors, and autocorrelated. But as residuals are not so high comparing to scale of inflation, it seems that models are possibly usable.

3 month horizon is quite short, generally speaking. The question is, what about long-term forecasting, e.g. half-yearly or yearly? They are equally, or even more interesting and relevant, e.g. for national budget planning.

Exercise 5 (0.75 points)

Perform forecasting for 6-month and yearly horizons, using: - both baselines - ETS - ARIMA

For the best model, plot the forecasts and perform residuals analysis.

Comment: - are there differences between models, compared to the 3-month forecasting? - how does the quality of forecasts change for longer horizons? - in your opinion, are those models useful at all for long-term forecasting?

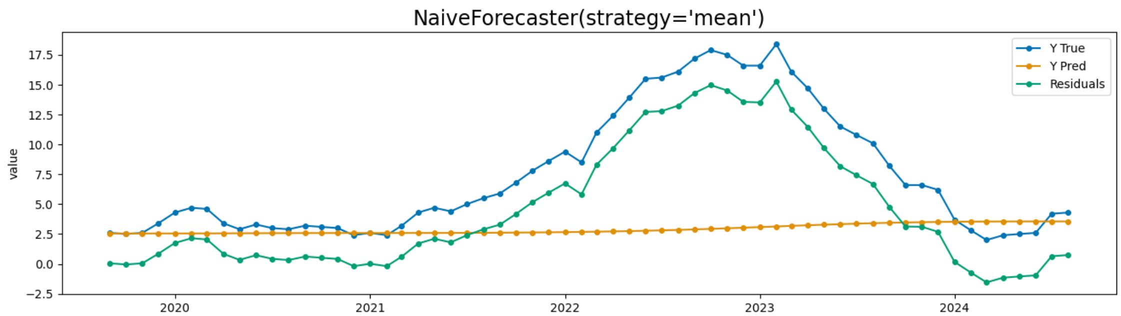
[18]:
```
# 1. Mean Forecasting
print("Testing Naive Forecaster with mean strategy")
mean_forecaster = NaiveForecaster(strategy="mean")
evaluate_model(mean_forecaster, data=y, horizon=6, plot_forecasts=True,␣
↪analyze_residuals=True)
```

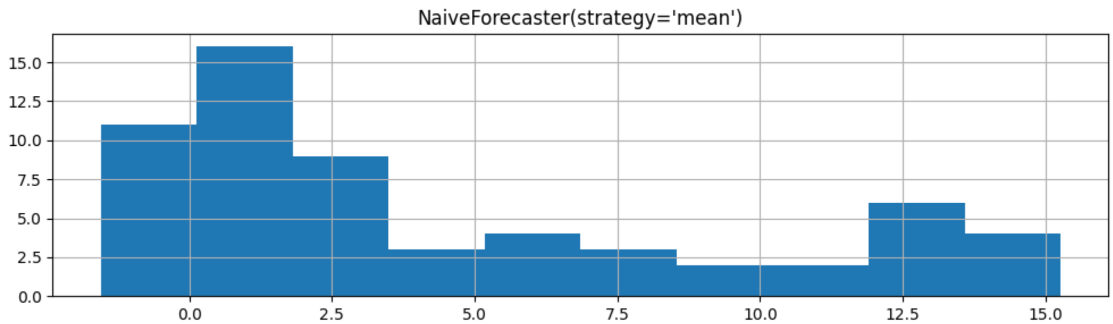
```
Testing Naive Forecaster with mean strategy
```

```
  0%|          | 0/55 [00:00<?, ?it/s]
```

```
Processing tasks:   0%|          | 0/55 [00:00<?, ?it/s]
```

```
MAE: 5.14
MASE: 15.15
```






```
Anderson-Darling test statistic: 3.4001356018721864
Ljung-Box test p-values: [7.04999234e-59]
```

```
<Figure size 640x480 with 0 Axes>
```

[19]:

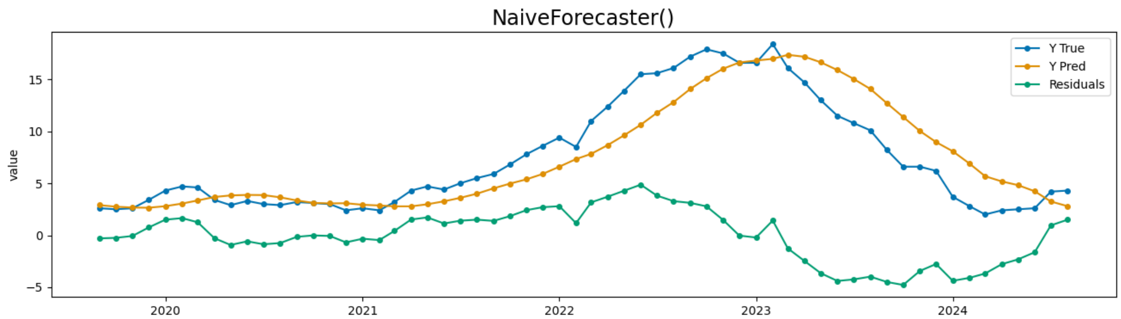
```
# 2. Testing Naive Forecaster with last known value strategy
print("\nTesting Naive Forecaster with last known value strategy")
last_forecaster = NaiveForecaster(strategy="last")
evaluate_model(last_forecaster, data=y, horizon=6, plot_forecasts=True,␣
↪analyze_residuals=True)
```

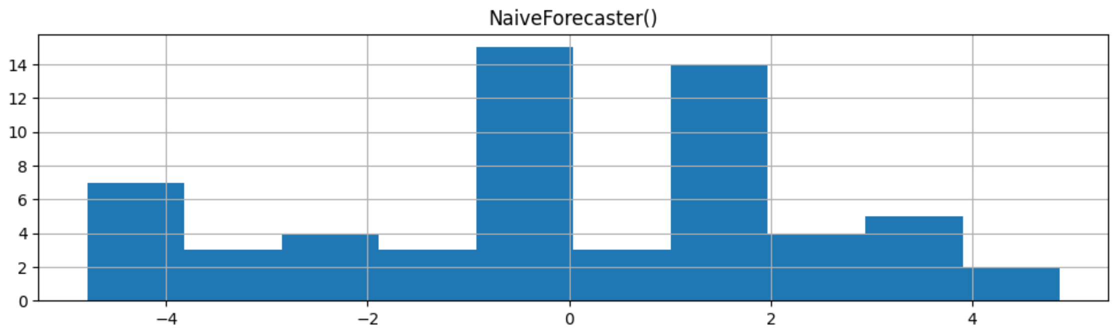
```
Testing Naive Forecaster with last known value strategy
```

```
  0%|          | 0/55 [00:00<?, ?it/s]
```

```
MAE: 2.15
MASE: 6.32
```

```
Processing tasks:   0%|          | 0/55 [00:00<?, ?it/s]
Processing tasks:   0%|          | 0/58 [00:12<?, ?it/s]
Processing tasks:   0%|          | 0/58 [00:11<?, ?it/s]
```






```
Anderson-Darling test statistic: 0.8145124068736465
Ljung-Box test p-values: [2.41668811e-43]
```

```
<Figure size 640x480 with 0 Axes>
```

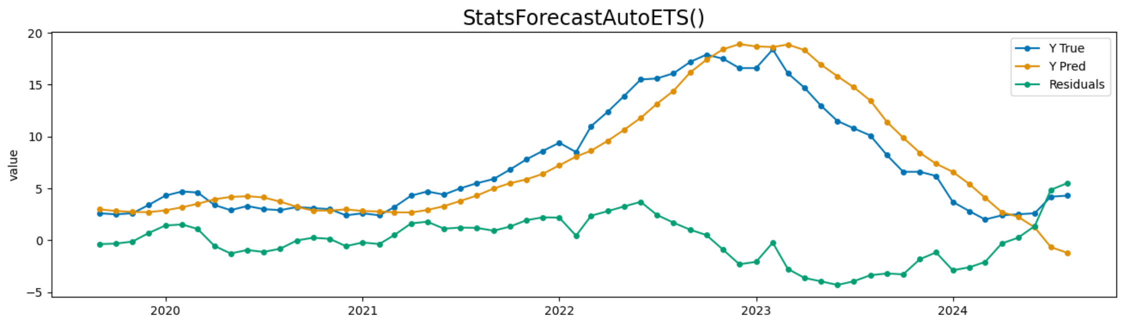
[20]:
```
# 3. Testing AutoETS Model with Damped Trend
print("Testing AutoETS with damped trend")
ets_forecaster = StatsForecastAutoETS(model="ZZZ") # "ZZD" denotes damped trend
evaluate_model(ets_forecaster, data=y, horizon=6, plot_forecasts=True,␣
 ↪analyze_residuals=True)
```

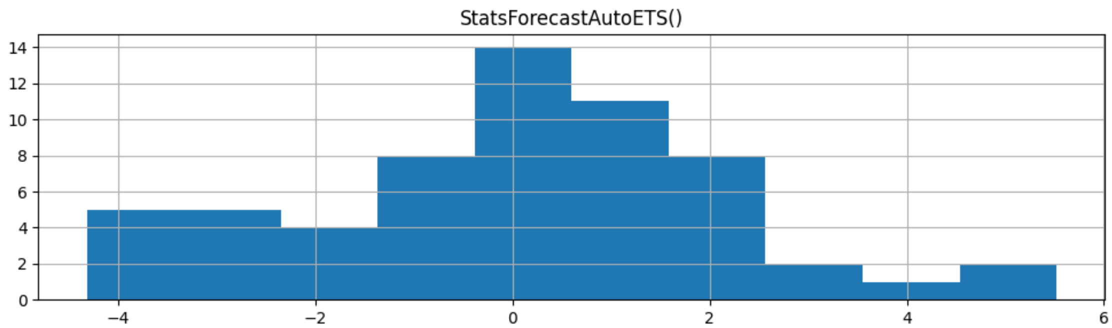
```
Testing AutoETS with damped trend
```

```
Processing tasks:   0%|          | 0/55 [00:00<?, ?it/s]
```

```
  0%|          | 0/55 [00:00<?, ?it/s]
```

```
MAE: 1.86
MASE: 5.49
```






```
Anderson-Darling test statistic: 0.32087462107819675
Ljung-Box test p-values: [6.42974956e-21]
```

```
<Figure size 640x480 with 0 Axes>
```

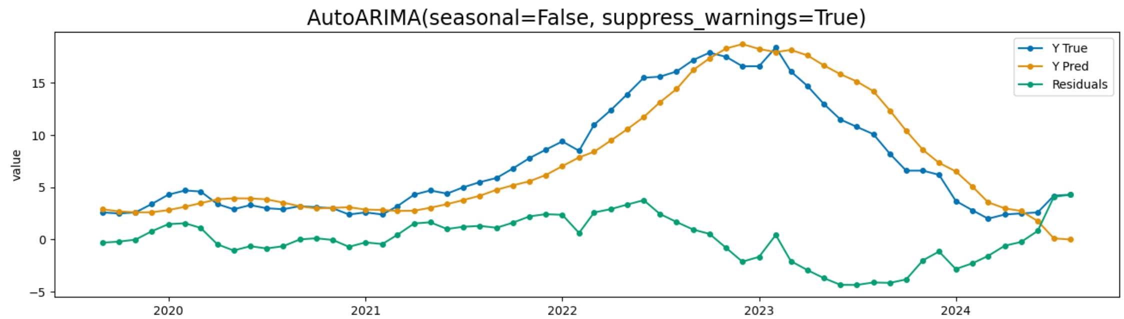
[21]:
```
# 4. Testing AutoARIMA Model without Seasonality
print("\nTesting AutoARIMA without seasonality")
arima_forecaster = AutoARIMA(suppress_warnings=True, seasonal=False)
evaluate_model(arima_forecaster, data=y, horizon=6, plot_forecasts=True,␣
  ↪analyze_residuals=True)
```

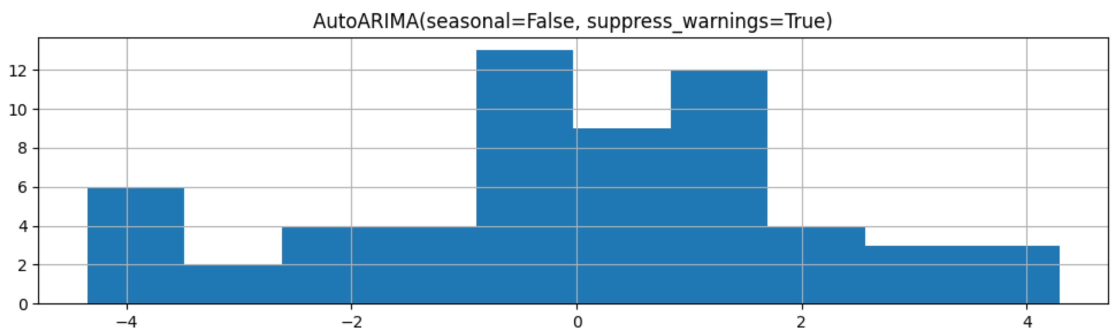
```
Testing AutoARIMA without seasonality
```

```
  0%|          | 0/55 [00:00<?, ?it/s]
```

```
Processing tasks:   0%|          | 0/55 [00:09<?, ?it/s]
```

```
MAE: 1.87
MASE: 5.51
```






```
Anderson-Darling test statistic: 0.36607020814739855
Ljung-Box test p-values: [9.83761592e-26]
```

```
<Figure size 640x480 with 0 Axes>
```

5 // commented

The same models were used. Results are worse than for horizon=3 So these models are rather not suitable for such long-term forecasts.

5.1 Forecasting network traffic

And now for something completely different. Network traffic forecasting is necessary for virtual machines (VMs) scaling, adding more servers to handle load in parallel. This is done more and more frequently by using ML models, based on time series forecasting, to scale more intelligently and avoid manually tweaking scaling rules. This is called predictive scaling, and is implemented by e.g. AWS, GCP, and Azure. There are also solutions for Kubernetes, both open source and proprietary. Time series forecasting allows lower latency and lower costs, automatically turning off machines when low demand is predicted.

Wikipedia and Google hosted Kaggle competition, where the goal was predicting the network traffic on particular Wikipedia pages. It's a really massive dataset, so we will operate on a simplified problem, where we have a total number of requests to the Wikipedia domain in millions.

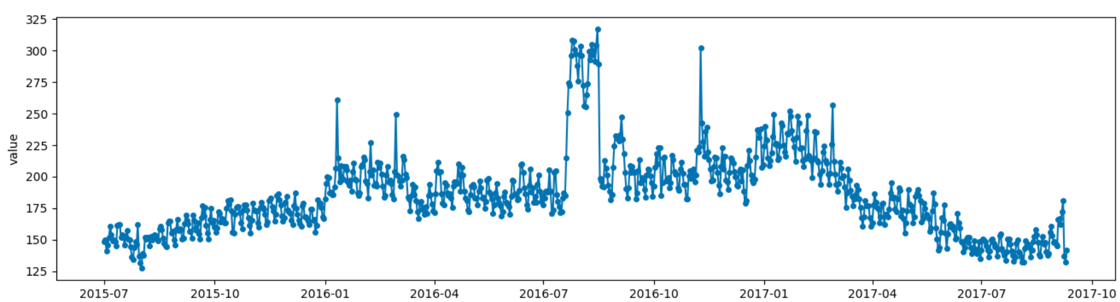
Typical characteristics of such tasks are: - short-term forecasting - high frequency - dynamically changing, noisy data (e.g. bot activity, web scraping) - frequent model retraining - high need for automatization, lack of manual model analysis

[22]:
```
y = pd.read_parquet("wikipedia_traffic.parquet")
y = y.set_index("date").to_period(freq="d")
plot_series(y)
y
```

[22]:

	value
2015-07-01	148.672476
2015-07-02	149.593840
2015-07-03	141.164198
2015-07-04	145.612937
2015-07-05	151.495372
…	…
2017-09-06	172.354146
2017-09-07	180.731284
2017-09-08	136.754670
2017-09-09	132.359512
2017-09-10	141.863949

[803 rows x 1 columns]



Exercise 6 (1 point)

For 1-day horizon, train models and evaluate them (similarly to the previous dataset, with 20% test data): - two baselines - ETS with damped trend - ARIMA (without seasonality) - SARIMA

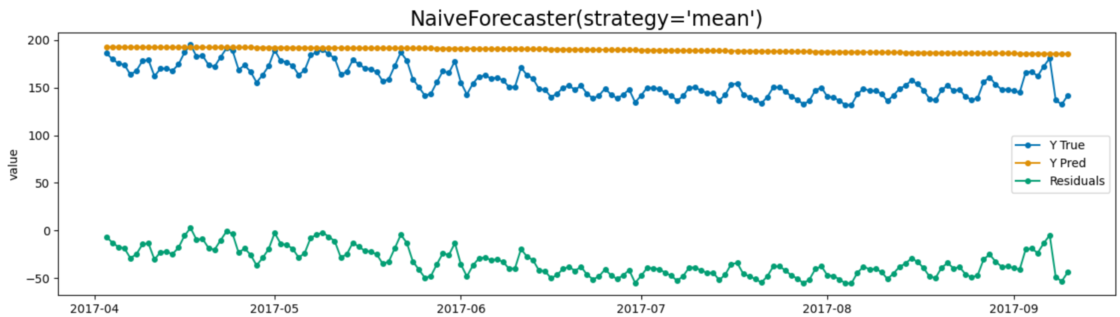
Comment: - based on those results, is there a seasonality here? - did you manage to outperform the baseline?

[23]:
```
# 1. Mean Forecasting
print("Testing Naive Forecaster with mean strategy")
mean_forecaster = NaiveForecaster(strategy="mean")
evaluate_model(mean_forecaster, data=y, horizon=3, plot_forecasts=True,␣
↪analyze_residuals=True)
```

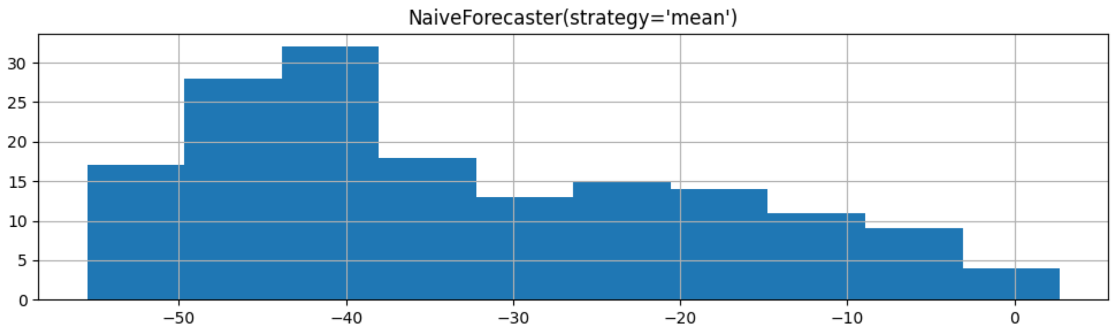
```
Testing Naive Forecaster with mean strategy
```

```
  0%|          | 0/159 [00:00<?, ?it/s]
```

```
MAE: 33.15
MASE: 3.93
```




```
<Figure size 640x480 with 0 Axes>
```




```
Anderson-Darling test statistic: 2.5602684030227465
Ljung-Box test p-values: [3.15628521e-123]
```

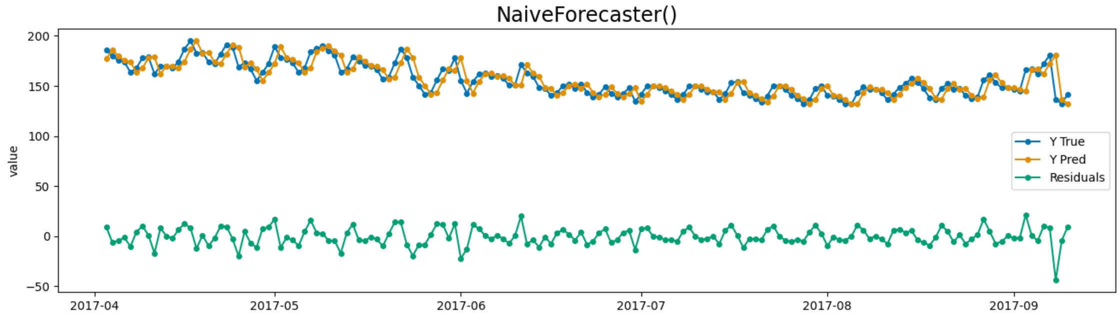
```
<Figure size 640x480 with 0 Axes>
```

[24]:
```
# 2. Testing Naive Forecaster with last known value strategy
print("\nTesting Naive Forecaster with last known value strategy")
last_forecaster = NaiveForecaster(strategy="last")
evaluate_model(last_forecaster, data=y, horizon=1, plot_forecasts=True,
  analyze_residuals=True)
```

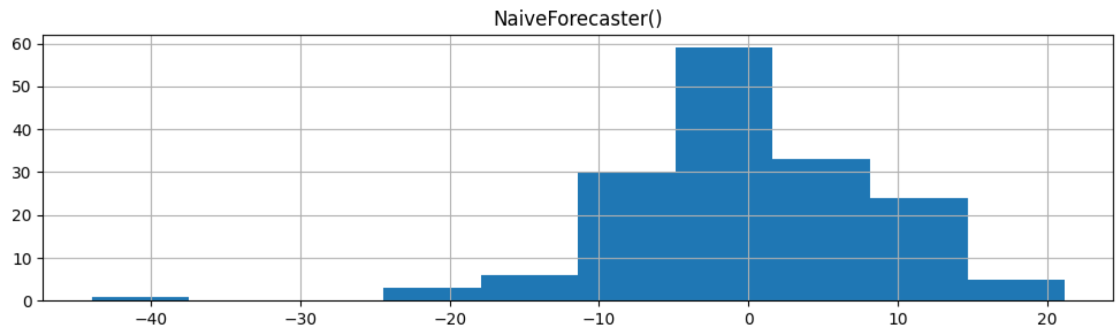
```
Testing Naive Forecaster with last known value strategy
```

```
  0%|          | 0/161 [00:00<?, ?it/s]
```

```
MAE: 6.63
MASE: 0.78
```




```
<Figure size 640x480 with 0 Axes>
```




```
Anderson-Darling test statistic: 0.7436284207647645
Ljung-Box test p-values: [1.36263891e-16]
```

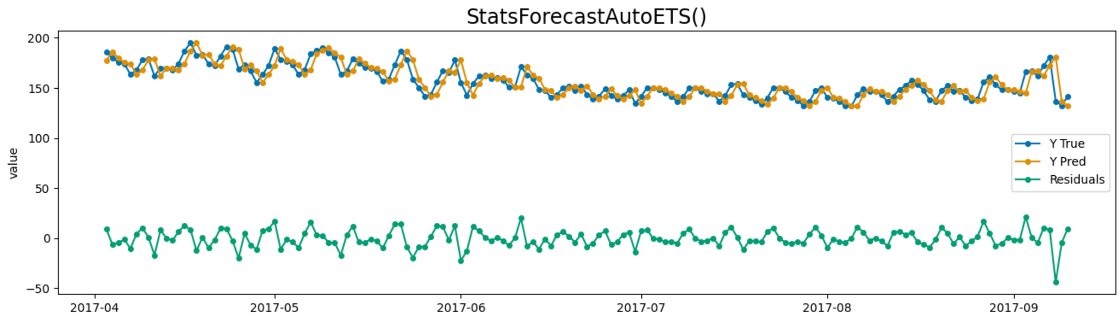
```
<Figure size 640x480 with 0 Axes>
```

[25]:

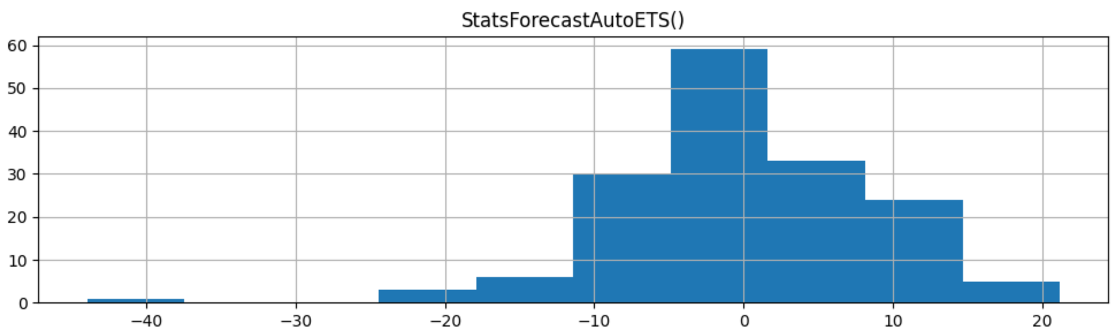
```python
# 3. Testing AutoETS Model with Damped Trend
print("Testing AutoETS with damped trend")
ets_forecaster = StatsForecastAutoETS(model="ZZZ") # "ZZD" denotes damped trend
evaluate_model(ets_forecaster, data=y, horizon=1, plot_forecasts=True,
  ↪analyze_residuals=True)
```

```
Testing AutoETS with damped trend
```

```
  0%|          | 0/161 [00:00<?, ?it/s]

Processing tasks:   0%|          | 0/55 [00:12<?, ?it/s]
Processing tasks:   0%|          | 0/159 [00:02<?, ?it/s]
Processing tasks:   0%|          | 0/161 [00:01<?, ?it/s]
Processing tasks:   0%|          | 0/161 [00:00<?, ?it/s]

MAE: 6.63
MASE: 0.78
```



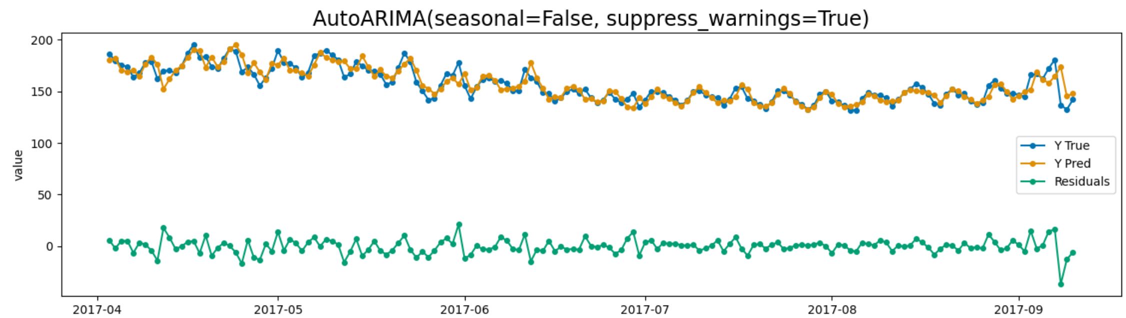

```
<Figure size 640x480 with 0 Axes>
```



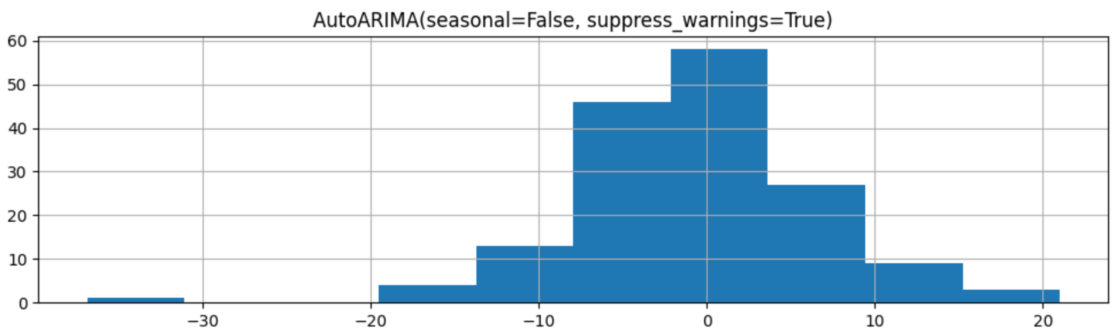

```
Anderson-Darling test statistic: 0.743565729676277
Ljung-Box test p-values: [1.35436936e-16]

<Figure size 640x480 with 0 Axes>
```

[26]:
```
# 4. Testing AutoARIMA Model without Seasonality
print("\nTesting AutoARIMA without seasonality")
arima_forecaster = AutoARIMA(suppress_warnings=True, seasonal=False)
evaluate_model(arima_forecaster, data=y, horizon=1, plot_forecasts=True,
    analyze_residuals=True)
```

```
Testing AutoARIMA without seasonality

Processing tasks:   0%|          | 0/161 [00:00<?, ?it/s]

  0%|          | 0/161 [00:00<?, ?it/s]

/home/xsalonx/miniconda3/envs/torch/lib/python3.10/site-
packages/pmdarima/arima/_auto_solvers.py:524: ModelFitWarning: Error fitting
ARIMA(4,1,3)(0,0,0)[0] intercept (if you do not want to see these warnings, run
with error_action="ignore").
  warnings.warn(warning_str, ModelFitWarning)

MAE: 5.22
MASE: 0.62
```




```
<Figure size 640x480 with 0 Axes>
```




```
Anderson-Darling test statistic: 1.173981589193005
Ljung-Box test p-values: [0.00583845]

<Figure size 640x480 with 0 Axes>
```

6 // Commented

Naive models were not outperformed. It seems there is not clear seasonality.

But maybe we can do better? This data is highly volatile, with high variance, which is particularly bad for ARIMA models. Let's apply the variance-stabilizing transform then. We have only positive values here, so there are no numerical problems.

Note that `Pipeline` from sktime is needed here (documentation), which will automatically invert the transformation during prediction. Sometimes models are evaluated on the transformed data, but we are generally interested in the forecasting quality on the data in its raw form. The goal of transformations is to make the training easier for the model.

Exercise 7 (0.5 points)

Create a pipeline, consisting of a transform object and AutoARIMA model (without seasonality). Try out the following transformations (documentation): - log - sqrt - Box-Cox

Comment, whether the result is better after the transformation or not.

[27]:
```
from tqdm import tqdm
from sktime.transformations.series.boxcox import BoxCoxTransformer
from sktime.transformations.series.exponent import ExponentTransformer
# from sktime.transformations.series.compose import TransformerPipeline



# Log Transformation Pipeline
log_pipeline =  ExponentTransformer(power=0.5) *
 ↪AutoARIMA(suppress_warnings=True, seasonal=False)

# Square Root Transformation Pipeline
sqrt_pipeline = ExponentTransformer(power=0.5) *
 ↪AutoARIMA(suppress_warnings=True, seasonal=False)

# Box-Cox Transformation Pipeline
boxcox_pipeline = BoxCoxTransformer() * AutoARIMA(suppress_warnings=True,
 ↪seasonal=False)

# Evaluate each pipeline
results = {}

for name, pipeline in tqdm([("Log", log_pipeline), ("Sqrt", sqrt_pipeline),
 ↪("Box-Cox", boxcox_pipeline)]):
    print(f"\nEvaluating pipeline with {name} transformation:")
    evaluate_model(pipeline, y, horizon=3, plot_forecasts=False,
 ↪analyze_residuals=True)
```

```
Evaluating pipeline with Log transformation:
```

```
  0%|          | 0/159 [00:00<?, ?it/s]
```

```
MAE: 6.81
MASE: 0.81
Anderson-Darling test statistic: 1.7967599919159625
Ljung-Box test p-values: [2.99962415e-09]

Evaluating pipeline with Sqrt transformation:
```

```
  0%|          | 0/159 [00:00<?, ?it/s]
```

```
MAE: 6.81
MASE: 0.81
Anderson-Darling test statistic: 1.7967599919159625
Ljung-Box test p-values: [2.99962415e-09]

Evaluating pipeline with Box-Cox transformation:
```

```
  0%|          | 0/159 [00:00<?, ?it/s]
```

```
100%|      | 3/3 [03:39<00:00, 73.21s/it]
```

```
MAE: 9.20
MASE: 1.09
Anderson-Darling test statistic: 0.8163278506456777
Ljung-Box test p-values: [7.19749612e-42]
```

None of methods improved model's performance

6.1 Sales forecasting

Arguably the most common application of time series forecasting is predicting sales, demand, costs etc., so all typical operational indicators of a company. Basically every company has to do this, therefore even basic software like Excel or PowerBI have built-in capabilities for time series forecasting.

We will focus on a task definitely vital for the Italian economy, i.e. the pasta sales. Dataset has been gathered by the Italian scientists for this paper. Data covers years 2014-2018, from 4 companies offering various pasta-based products. They also contain data about promotions for particular products. There are also missing values, which must be imputed.

Typical characteristics of this type of data are: - positive trend, smaller or larger (changing in time) - strong seasonality, often more than one - highly sensitive to recurring events, e.g. weekends or holidays - large outliers, often related to events - relatively low frequency, daily or less frequent - often long forecasting horizons, e.g. monthly, quarterly, yearly - rich exogenous variables

Exercise 8 (1 point)

1. Read the data from `"italian_pasta.csv"` file
2. Select columns from company B1 (they have `"B1"` in their name) and `"DATE"` column.
3. Create the `value` column with total pasta sales, i.e. sum of columns with `"QTY"` in name.
4. Create the `num_promos` column with total number of promotions, i.e. sum of columns with `"PROMO"` in name.
5. Leave only columns `"DATE"`, `"value"` and `"num_promos"`.
6. Create index with type `datetime`:
    - change type of `"DATE"` colum to `datetime`
    - set its frequency as daily, `"d"`
    - set it as index
7. Split the data into:
    - `y` variable, `pd.Series` created from the `"value"` column, our main time series values
    - `X` variable, `pd.Series` created from the `"num_promos"` column, exogenous variables
8. Impute the missing values in exogenous variables with zeros, assuming that by default there are no promotions.
9. Plot the `y` time series. Remember to set the appropriate title.

[28]:
```
# Step 1: Load data
y = pd.read_csv("italian_pasta.csv")

# Step 2: Select Company B1 columns
b1_qty_columns = [col for col in y.columns if col.startswith("QTY_B1")]
b1_promo_columns = [col for col in y.columns if col.startswith("PROMO_B1")]

# Keep only relevant columns along with the "DATE" column
df_b1 = y[["DATE"] + b1_qty_columns + b1_promo_columns].copy()

# Step 3: Aggregate data
df_b1["value"] = df_b1[b1_qty_columns].sum(axis=1)  # Total sales for Company B1
df_b1["num_promos"] = df_b1[b1_promo_columns].sum(axis=1)  # Total promotions␣
↪for Company B1

# Step 4: Format the data
df_b1["DATE"] = pd.to_datetime(df_b1["DATE"])
df_b1.set_index("DATE", inplace=True)
df_b1 = df_b1.asfreq("D")  # Set frequency to daily

# Step 5: Create main variables for analysis
y = df_b1["value"].sort_index()
```

```python
X = df_b1["num_promos"].sort_index()

# Step 6: Impute missing values in X (promotions)
# Fill missing promotions with zero, assuming no promotions
X = X.fillna(0)
y = y.fillna(0)

df_b1.head()
```

[28]:

```
            QTY_B1_1  QTY_B1_2  QTY_B1_3  QTY_B1_4  QTY_B1_5  QTY_B1_6  \
DATE
2014-01-02       7.0       3.0       0.0       2.0       3.0       1.0
2014-01-03       5.0       0.0       0.0       6.0       9.0       1.0
2014-01-04       9.0       7.0       2.0       1.0       5.0       2.0
2014-01-05       5.0       1.0       2.0       2.0       3.0       0.0
2014-01-06       1.0       0.0       1.0       0.0       1.0       0.0

            QTY_B1_7  QTY_B1_8  QTY_B1_9  QTY_B1_10  …  PROMO_B1_35  \
DATE                                                 …
2014-01-02       0.0       4.0       0.0        0.0  …          0.0
2014-01-03       2.0       4.0       0.0        1.0  …          0.0
2014-01-04       0.0       6.0       4.0        5.0  …          0.0
2014-01-05       1.0       4.0       5.0        1.0  …          0.0
2014-01-06       0.0       2.0       1.0        1.0  …          0.0

            PROMO_B1_36  PROMO_B1_37  PROMO_B1_38  PROMO_B1_39  PROMO_B1_40  \
DATE
2014-01-02          0.0          0.0          0.0          0.0          0.0
2014-01-03          0.0          0.0          0.0          0.0          0.0
2014-01-04          0.0          0.0          0.0          0.0          0.0
2014-01-05          0.0          0.0          0.0          0.0          0.0
2014-01-06          0.0          0.0          0.0          0.0          0.0

            PROMO_B1_41  PROMO_B1_42  value  num_promos
DATE
2014-01-02          0.0          0.0  101.0         0.0
2014-01-03          0.0          0.0  136.0         0.0
2014-01-04          0.0          0.0  162.0         0.0
2014-01-05          0.0          0.0  106.0         0.0
2014-01-06          0.0          0.0   47.0         0.0

[5 rows x 86 columns]
```

[29]:

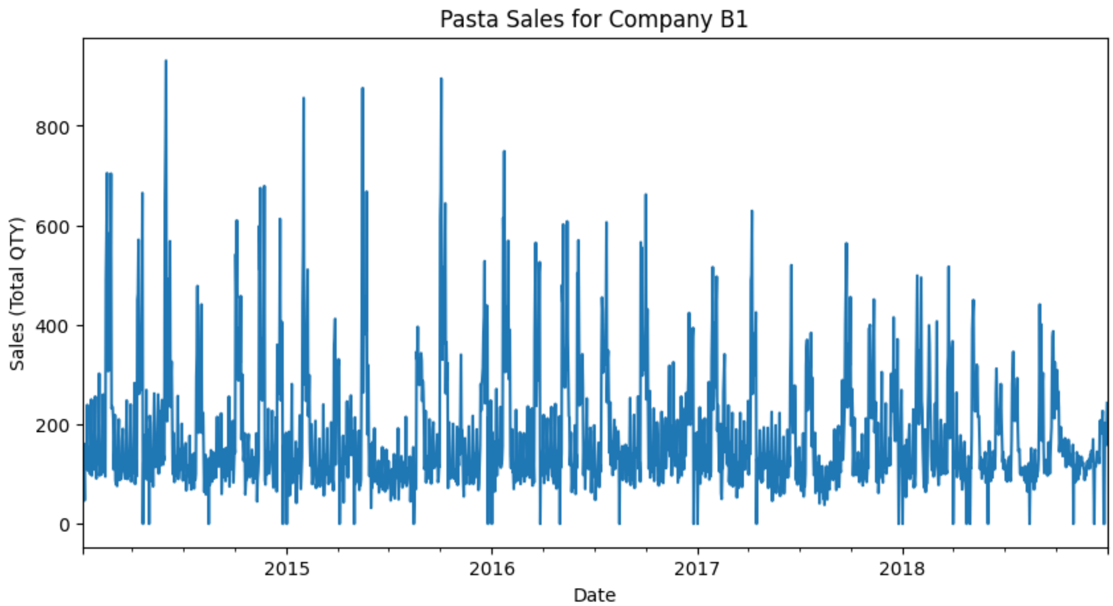
```python
print('y = ', y)
print()
print('X = ', X)
```

```
y =  DATE
```

```
2014-01-02    101.0
2014-01-03    136.0
2014-01-04    162.0
2014-01-05    106.0
2014-01-06     47.0
              …
2018-12-27    203.0
2018-12-28    192.0
2018-12-29    158.0
2018-12-30    182.0
2018-12-31    243.0
Freq: D, Name: value, Length: 1825, dtype: float64

X =  DATE
2014-01-02     0.0
2014-01-03     0.0
2014-01-04     0.0
2014-01-05     0.0
2014-01-06     0.0
              …
2018-12-27    19.0
2018-12-28    21.0
2018-12-29    18.0
2018-12-30    13.0
2018-12-31    16.0
Freq: D, Name: num_promos, Length: 1825, dtype: float64
```

[30]:

```
plt.figure(figsize=(10, 5))
y.plot(title="Pasta Sales for Company B1", xlabel="Date", ylabel="Sales (Total␣
 ↪QTY)")
plt.show()
```



We are interested in long-term forecasting. We assume that our client, an italian pasta maker, has the historical data from years 2014-2017 and wants to forecast the sales for 2018. Such information is required e.g. to make contracts for long-term supply of raw materials and next year production plans. From ML perspective this hard, since there is only a single temporal train-test split with long horizon, instead of expanding window, but it's faster.

We will use the `evaluate_pasta_sales_model` function for evaluation.

Exercise 9 (1 point)

Implement the missing parts of the evaluation function: 1. Split `y` into training and testing set with time split. Test set starts at `2018-01-01`. 2. If user passes `X`, split it in the same way. 3. Impute the missing values in `y`, using `Imputer` from sktime (documentation) with `ffill` strategy (copy last known value). 4. Train the model (remember to pass `X`) and perform prediction. 5. Evaluate it using MAE and MASE functions (documentation). Print the results rounded to 2 decimal places. 6. Copy the code for `analyze_residuals` from exercise 3.

[31]:
```
from typing import Optional

import numpy as np
import pandas as pd
from sktime.performance_metrics.forecasting import (
    mean_absolute_scaled_error,
    mean_absolute_error,
    mean_absolute_percentage_error,
)
from sktime.transformations.series.impute import Imputer
from sktime.utils.plotting import plot_series
```

```python
import matplotlib.pyplot as plt

def evaluate_pasta_sales_model(
    model,
    y: pd.Series,
    X: Optional[np.ndarray] = None,
    plot_forecasts: bool = False,
    analyze_residuals: bool = False,
    plot_residuals_hist = True,
) -> None:
    # Step 1: Split data into training and testing sets
    train_size = int(len(y) * 0.8)
    y_train, y_test = y.iloc[:train_size], y.iloc[train_size:]

    if X is not None:
        X_train, X_test = X[:train_size], X[train_size:]
    else:
        X_train = None
        X_test = None

    # Step 2: Impute missing values
    imputer = Imputer(method="mean")
    y_train = imputer.fit_transform(y_train)
    if X_train is not None:
        X_train = imputer.fit_transform(X_train)
        X_test = imputer.transform(X_test)

    # Step 3: Train the model and make predictions
    model.fit(y_train, X=X_train)
    y_pred = model.predict(fh=np.arange(1, len(y_test) + 1), X=X_test)

    # Step 4: Calculate evaluation metrics
    mae = mean_absolute_error(y_test, y_pred)
    mape = mean_absolute_percentage_error(y_test, y_pred)
    mase = mean_absolute_scaled_error(y_test, y_pred, y_train=y_train)

    print(f"MAE: {mae:.2f}")
    print(f"MAPE: {mape:.2f}")
    print(f"MASE: {mase:.2f}")

    y_true = y.loc[y_pred.index]
    residuals = y_true - y_pred

    # Step 5: Plot forecasts if requested
    if plot_forecasts:
        plot_series(y_test, y_pred, residuals, labels=["y_test", "y_pred",␣
 ↪"residuals"])
```

```
        plt.title("Actual vs Predicted Pasta Sales")
        plt.show()
        plt.clf()


    # Step 6: Analyze residuals if requested
    if analyze_residuals:
        if plot_residuals_hist:
            plt.figure()
            plt.hist(residuals)
            plt.title('Residuals histogram')

        # Anderson-Darling test for normality of residuals
        anderson_result = anderson(residuals.values.reshape(-1))
        print("Anderson-Darling test statistic:", anderson_result.statistic)

        # Ljung-Box test for autocorrelation in residuals
        ljung_box_result = acorr_ljungbox(residuals, lags=[10], return_df=True)
        print("Ljung-Box test p-values:", ljung_box_result["lb_pvalue"].values)
```

Exercise 10 (1.5 points)

Perform the forecasting using the following models: - two baselines - ETS with damped trend - ARIMA - SARIMA with 30-day seasonality - ARIMAX - SARIMAX with 30-day seasonality

For the best model also try the log, sqrt and Box-Cox transformations.

For the final model plot the forecasts and perform residuals analysis.

Comment: - did you outperform the baseline? - does the final model use seasonality and/or exogenous variables (data about promotions)? - was it worth it to use the variance-stabilizing transformation? - comment on the general behavior of the model on the test set, based on the forecast plot - is the model unbiased (normally distributed residuals with zero mean), without autocorrelation, or can this be improved?

[ ]:
```
from sktime.forecasting.arima import AutoARIMA
from sktime.forecasting.ets import AutoETS
from sktime.forecasting.naive import NaiveForecaster
from sktime.forecasting.naive import NaiveForecaster
from sktime.forecasting.ets import AutoETS
from sktime.forecasting.arima import AutoARIMA

# Define a dictionary with each model configuration
models = {
    "Naive (Last)": NaiveForecaster(strategy="last"),
    "Naive (Mean)": NaiveForecaster(strategy="mean"),
    "ETS (Damped Trend)": AutoETS(damped_trend=True, trend="additive"),
    "ARIMA": AutoARIMA(suppress_warnings=True, seasonal=False),
    "SARIMA (30-Day)": AutoARIMA(suppress_warnings=True, seasonal=True, sp=30),
```

```
    "ARIMAX": AutoARIMA(suppress_warnings=True, seasonal=False),  # Exogenous
    "SARIMAX (30-Day)": AutoARIMA(suppress_warnings=True, seasonal=True,␣
 ↪sp=30),  # Exogenous
}

# Iterate over models, calling evaluate_pasta_sales_model for each
for model_name, model in tqdm(models.items()):
    print(f"\nEvaluating model: {model_name}")
    if model_name == 'ARIMAX' or model_name == "SARIMAX (30-Day)":
        pass_X = X
    else:
        pass_X = None

    evaluate_pasta_sales_model(model, y=y, X=pass_X, plot_forecasts=False,␣
 ↪analyze_residuals=True, plot_residuals_hist=False)
```

```
Evaluating model: Naive (Last)
MAE: 59.81
MAPE: 167805355978736280.00
MASE: 0.86
Anderson-Darling test statistic: 16.554517894292644
Ljung-Box test p-values: [9.2287584e-69]

Evaluating model: Naive (Mean)
MAE: 72.60
MAPE: 223284396087197680.00
MASE: 1.05
Anderson-Darling test statistic: 16.554517894292644
Ljung-Box test p-values: [9.2287584e-69]

Evaluating model: ETS (Damped Trend)
MAE: 70.39
MAPE: 216913872102181240.00
MASE: 1.01
Anderson-Darling test statistic: 16.55452363054667
Ljung-Box test p-values: [9.22831812e-69]

Evaluating model: ARIMA


MAE: 72.29
MAPE: 219133535317250120.00
MASE: 1.04
Anderson-Darling test statistic: 16.406380129347554
Ljung-Box test p-values: [9.01944493e-69]
```

```
Evaluating model: SARIMA (30-Day)
```

Exogenous variables can be expanded with feature engineering. For example, the behavior of clients is quite different during weekends and holidays. Typically sales rise quite sharply before and after days when stores are closed, and falls to exactly zero when they have to be closed.

Exercise 11 (0.75 points)

1. Create a list of variables for holidays using `HolidayFeatures` (documentation):
   - use `country_holidays` function from the holidays library
   - remember that we are processing italian data, with country identifier `"IT"`
   - include weekends as holidays
   - create a single variable "is there a holiday" (`return_dummies` and `return_indicator` options)
2. Add those features to our exogenous variables `X`. Use `pd.merge` function, `left_index` and `right_index` options may be useful.
3. Train the ARIMAX model (or SARIMAX, if you detected seasonality before). Use the best transformation from the previous exercise.
4. Comment on the results, and compare them to the previous ones.

// comment here

[ ]:

```
```